ATARI.RSC
The Atari Developer's Resource

Vol. V, Issue 2
December 1992

Atari Falcon030 Debuts

Mike Fulton

The Atari Falcon030 made its official debut at the annual Dusseldorf Atari Messe (fair) August 21-23, ending months of speculation and rumors. The Atari Falcon030 is the first in a new line of TOS-compatible machines. Here are the basic specifications of the machine:

* 16MHz 68030 CPU
* 32MHz Motorola DSP56K Digital Signal Processor (20 mips)
* 16-bit Stereo DMA Sound Record/Playback, with up to 50kHz sampling rate
* 16-bit True Color video modes with 65536 colors
* Palette-based video modes with 2, 4, 16, or 256 colors at once, from a palette of 262,144 possible colors (or 4096 colors in ST-compatible video modes)
* Overscan video modes
* Up to 768 x 480 resolution (w/overscan)
* GENLOCK Overlay capabilities in true color video modes
* Enhanced GEM Desktop with Color Icons
* Enhanced GEM AES with 3D object types, hierarchical drop-down menus, popup-menus, and 3D window gadgets.
* 1, 4, or 14mb RAM
* Optional internal 2.5" IDE Hard Disk Drive
* Uses VGA/MultiSync, Atari ST-compatible monochrome or color monitor, or television
* MIDI In/Out ports
* Parallel port, with new signals added that make it easier to connect devices like scanners, etc.
* Localtalk™LAN port
* Serial port, Modem 1 port now uses SCC chip (for higher transfer rates) instead of 68901 MFP chip.
* Internal CPU expansion bus
* STE-compatible hardware video scrolling
* STE-compatible joystick ports
* SCSI 2 port for hard disk drives, scanners, tape drives, etc.

And of course, there are many other new features, but it would take too long to list everything.

Approximately 25 machines were on display at the show, and virtually all of them were purchased by excited developers at the end of the show. Additional developer units should be available soon. Please contact your developer administrator for more information.

About twenty developers had received machines in the months before the show, and many of them had demonstrations of new software packages at the show, including D2D Systems, who demonstrated a direct to hard disk, 16-bit stereo digital recording program, and Hisoft, who was showing a true color paint program, True Paint.

Atari Falcon030 U.S. debut at Boston Computer Society

On Sept. 23, the Atari Falcon030 made its official debut in the U.S. at the monthly meeting of the Boston Computer Society. Atari's President Sam Tramiel introduced the machine and along with Richard Miller, Bill Rehbock, and James Grunke, discussed the machines capabilities and the targeted position in the computer market.

There has been much interest in the press as newspapers across the country featured articles about the new machine and interviews with Sam Tramiel. The syndicated cable television show PCTV also featured an interview and demonstration with Bill Rehbock showing off some of the Falcon030's sophisticated audio capabilities.

Inside This Issue

ATARI.RSC The Resource File

TECHNICAL DIRECTOR
Bill Rehbock (408) 745-2082
GEnie: B.REHBOCK
CIS: 75300,1606
Internet:
75300.1606@compuserve.com

DEVELOPER TECHNICAL SUPPORT
J. Patton (408) 745-2135
GEnie: ATARIDEV
CIS: 70007,1072
Internet:
70007.1072@compuserve.com

Mike Fulton (408) 745-8821
GEnie: MIKE-FULTON
CIS: 75300,1141
Internet:
75300.1141@compuserve.com

Fax: (408) 745-2094

DEVELOPER ADMINISTRATOR
Gail Bacani (408) 745-2022

Atari Computer Corp.
1196 Borregas Ave.
Sunnyvale, CA 94089-1302

ATARI.RSC edited
by Mike Fulton

ATARI.RSC was created using Pagestream v2.2 on the Atari TT030 computer & Atari TTM194 monochrome monitor, and was printed using CompoScript on the Atari SLM605 Laser Printer.

Editor's Notes

Mike Fulton

Until now, ATARI.RSC has been primarily aimed at the developers in North America, and past issues have featured many excellent articles from developers in the U.S.A. and Canada. Now the focus of ATARI.RSC has been expanded to Atari developers throughout the world, and we'd like to see articles from other countries as well.

Articles must be submitted in English (don't worry if your English isn't very good, we can fix that), and in RAW ASCII text format. You can submit articles to my attention either by mail (please include both a printout and a floppy disk) or through EMAIL (addresses shown at left.) Please include a return address, telephone number, and EMAIL address so that I can get back in touch with you.

Article subjects can be nearly anything you like, but it should be something that most other developers will find interesting. Also, programming methods and any program listings should conform to Atari's system programming and user interface guidelines.

Documentation Correction

The GEM VDI manual documentation for the ***vq_extnd()*** (Extended Inquire) function contains the following text on page 8-2:

```
intout[5] -- Lookup table supported.

        0 = table supported
        1 = table not supported
```

This is incorrect. The values are reversed. The correction is:

```
intout[5] -- Lookup table supported.

        0 = table not supported
        1 = table supported
```

Table Supported indicates video hardware that uses hardware color registers to hold the RGB value, and the pixel value itself indicates which color register the RGB value comes from (like the ST/STe/TT machines). ***Table Not Supported*** indicates either a monochrome video mode or a true-color video hardware where the pixel value itself indicates the RGB value to be displayed.

SpeedoGDOS

The new SpeedoGDOS, along with new device drivers and the CHARTER Speedo font family, is now available online in the GDOS library of the ATARI.RSC Developer Roundtable on GEnie. SpeedoGDOS is a new version of GDOS that uses Bitstream Speedo format outline fonts. The GDOS library is private; to request access, please leave email to MIKE-FULTON or ATARIDEV.

One new feature of the new printer drivers that come with SpeedoGDOS is that they now recognize the concept of an unprintable margin around the edges of your page. For example, most printers can only print 8" across the page, but your paper is actually 8.5" wide, so there's 0.5" that cannot be printed. The new drivers allow for this by placing the (0,0) coordinate of your page at the top left edge of the paper, outside the printable area, rather than at the start of the printable area, which may have been, for example, 0.25" in from the left and 0.125" down from the top.

The main idea behind the change is that if you set a 1.5" margin in your word processor, you want to get a margin based on the actual paper, like you'd expect, rather than based on the printable area of the printer. Now you'll be able to do that.

(continued on page 5)

GEM VDI & True Color Video

Mike Fulton

In palette-based video modes, a pixel value represents a hardware color register. Each hardware color register contains an RGB color value which determines the actual color used to display the pixel. If you change the RGB values contained in a color register, all pixels in the display which contain the value corresponding to that register will change color. The built-in video of the ST/STe/TT series is palette-based.

With true color video, the value of a pixel directly represents the actual RGB value that will be displayed. Each pixel's color is set independently of all other pixels and changing its color is done by changing the pixel value itself. The Atari Falcon030 features both palette-based and true color video capabilities.

Detecting True Color Video

If you request the extended inquire values from the ***vq_extnd()*** function for a screen workstation, the value returned in *intout[5]* specifies if a color lookup table is supported. (See the documentation correction in this newsletter.) If it's zero, it means the device does not support a color lookup table, and that the pixel values themselves directly represent the displayed color. This means you are either in a true color video mode, or that you are in a monochrome mode. To determine which, check the number of bitplanes (*intout[4]* from ***vq_extnd()***), if greater than one, then you're in a true color video mode, and if it's equal to one, then you're in a monochrome video mode.

What's Different?

From a programmer's viewpoint, for the most part, GEM VDI works with true color video modes in the same way that it does in palette-based video modes. However, there are some inevitable differences which cannot be avoided. Some GEM VDI functions take different parameters or behave a bit differently in true color video modes. The information below details these differences, and should be used in conjunction with your GEM VDI manual.

The *intout[13]* value from ***v_opnwk()*** or ***v_opnvwk()*** returns the number of pre-defined colors, or pens, that are available. For palette-based video, this indicates how many colors can be shown at once (without resorting to doing things like using interrupts to change the color registers). If you change the RGB value of a pen using ***vs_color()***, then any pixels on screen which were drawn with that pen will change to the new color.

For true color video modes, *intout[13]* indicates how many pre-defined pens there are, but not how many colors can be displayed on screen at once. The pens contain RGB values that will be used to draw whenever you use VDI functions like ***v_pline()*** or ***v_bar()***, but they don't necessarily have a direct relationship to what's already on screen like they do in palette-based video modes.

In other words, changing a pen's color with ***vs_color()*** does not affect anything you've already drawn with that pen. For example, if you draw a circle with pen 12, then change pen 12's RGB value with ***vs_color()***, and then draw a square with pen 12, you will end up with the circle in pen 12's original RGB color and the square in pen 12's new RGB color. One result of this behavior is that you cannot do color cycling by simply changing the pen's color values with ***vs_color()***.

vro_cpyfm()

```
vro_cpyfm( handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB );
WORD handle;
WORD wr_mode;
WORD pxyarray[8];
MFDB *psrcMFDB;
MFDB *pdesMFDB;
```

The *wr_mode* parameter indicates the write mode to use in copying a rectangular area from one raster form to another. In both true color and palette-based modes, this is a bit-level operation. Since the bits for a pixel value mean something different in true color modes, the onscreen results of the same logic operation can be different. They are easily predictable, however.

Some programs use ***vro_cpyfm()*** with write mode 0 to clear areas of the screen or to clear an off-screen buffer. This sets all bits in the destination rectangle to zero, regardless of the original value. In palette-based modes, this has the effect of causing the pixels within that rectangle to be displayed using Pen 0, which is the background color, and which can be any RGB value. However, in true color video modes, setting all the bits to zero causes the pixels to be displayed as black, no matter what RGB value pen 0 is set to, since the values for red, green, and blue are all zero. In cases where the background color isn't supposed to be black, say for example in an animation playback, the results aren't going to be what was intended.

So how do you get around this? You can use ***vrt_cpyfm()*** and a dummy raster form instead of ***vro_cpyfm()***.

The ***vrt_cpyfm()*** function blits from a single-plane (1 bit per pixel) raster to a multi-plane raster. You specify two VDI pens to be used for drawing pixels in the destination raster: one for the pixels with 1's in the source raster, and one for the pixels with 0's. If you specify the same pen for both, then the whole rectangle will be drawn with the same pen, and it doesn't even really matter what the source MFDB points to (the easiest thing to do is point it at the same memory as the destination raster, but set the MFDB to one plane). If you wanted to set a rectangular area of the raster to the background color, then you could just use ***vrt_cpyfm()*** with zero for both pens. Unlike other VDI functions that use pens, like ***v_bar()*** or ***vr_recfl()***, the ***vrt_cpyfm()*** function can work with offscreen buffers with no significant extra effort.

Another thing to watch out for is the way other logic operations react. With palette based modes, if one did an OR blit operation with a source raster containing pixel values of 1 and a destination raster containing pixel values of 4, you'd end up with the destination rectangle containing pixel values of 5 (1 OR 4 = 5) which could be the same RGB color as either the pixel value 1 or pixel value 4, or even something completely different and apparently unrelated as far as RGB colors are concerned. In true color modes, if you did an OR blit operation with a source raster containing red pixels and a destination raster containing blue pixels, you'd end up with the destination raster containing magenta (red + blue = magenta) pixels. (And if you think about it a bit, you'll see that this sort of thing could be very useful.)

What ***vro_cpyfm()*** does to the raster memory hasn't really changed; the bits are affected in the same way as before, but now the results may not always mean the same thing they used to. While the replace write mode hasn't changed, other write modes with ***vro_cpyfm()*** may also give different results from palette-based modes. Some experimentation may be required to obtain the desired results.

vswr_mode()
vrt_cpyfm()

```
vswr_mode( handle, mode )
WORD handle;
WORD mode;

vrt_cpyfm( handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB,
           color_index )
WORD handle;
WORD wr_mode;
WORD pxyarray[8];
MFDB *psrcMFDB;
MFDB *pdesMFDB;
WORD color_index[2];
```

The ***vswr_mode()*** function sets the write mode logic that GEM VDI will use in drawing pixels to the screen for all calls except ***vro_cpyfm()*** and ***vrt_cpyfm()***, both of which use their own write mode parameters. Since the write modes for ***vrt_cpyfm()*** are the same as for ***vswr_mode()***, the information below refers to both ***vswr_mode()*** and ***vrt_cpyfm()***.

```
    mode = 0 Replace mode
           1 Transparent mode
           2 eXclusive OR (XOR) mode
           3 Reverse Transparent
```

As with ***vro_cpyfm()***, the differences with ***vswr_mode()*** arise from the fact that the logic of the write mode works at the bit level.

In palette-based modes, the write modes work as follows: Replace mode simply copies bits from the new pixel value into the destination pixel, completely replacing the old value with the new value. XOR mode does an exclusive-OR logic operation between the new pixel value and the destination pixel's existing value, and sets the pixel to the resulting value. Transparent mode affects pixels only when they are not being drawn with pen zero. Reverse-Transparent mode only affects pixels that are supposed to be drawn using pen zero, except it uses the drawing color to draw those pixels instead.

Figure #1 shows an example of each type of write mode. The circle was drawn first using replace mode, then the box was drawn on top, using a different write mode each time.

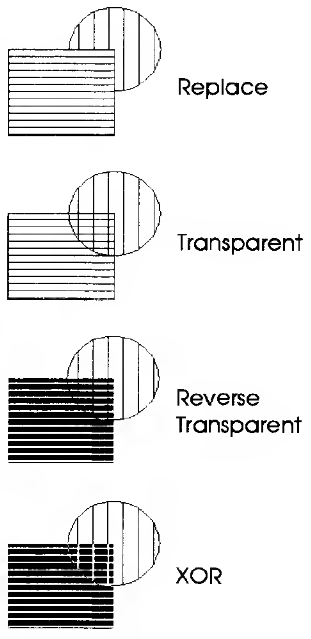


Figure #1

If the fill color is set to pen four, then in replace mode, the box outline and hatch pattern will be drawn using pen four, and the rest of the inside of the box will be drawn with pen zero so that whatever was previously underneath will be completely replaced. In transparent mode the rest of the inside of the box won't be drawn at all, and whatever was previously underneath will still be there. In reverse transparent mode, the outline and hatch pattern won't be drawn at all, but the rest of the inside of the box will be drawn, using pen four (the fill color) instead of pen zero. In XOR mode, the box outline and hatch pattern will be drawn in whatever is the result of pen 4 XOR the existing pixel values, and the rest of the inside of the box will be drawn in the result of pen 0 XOR the existing pixel values.

Using XOR mode can be a little different in true color video modes. A thing sometimes done in palette-based modes is to write some object to the screen against the background color, then write it again using XOR mode to erase it. In true color modes there are two ways this can fail unless the background is pure-black.

If XOR mode is used to draw the object both times, then the it will be drawn in the wrong colors if the background isn't pure black, but will be erased as expected when drawn the second time. If XOR mode isn't used to draw the object in the first place, but replace mode is used instead, then the object will be drawn in the correct colors, but it will not be erased properly when drawn the second time. Instead of the background color reappearing, you'll get black instead because any number XOR'ed with itself is zero, and in true color, a pixel value of zero means black.

Another difference of XOR mode is that when you XOR a pixel in a palette-based mode, the result is a different pixel value, which will have its own RGB value that may not have any meaningful relationship to the original pixel's RGB value and the XOR operation. In true color mode, if you XOR a pixel, the resulting color is both easily predictable and meaningful. For example, if you XOR any RGB value with a pure white

RGB value (0xffff for 16-bit or 0x00ffffff for 24/32 bit) then you'll get the opposite color. For example, if you XOR a red pixel with 0xffff and you get a cyan (blue+green) pixel, which is the opposite color on a color wheel. If you XOR a yellow (red+green) pixel with 0xffff, you get blue.

v_get_pixel()

```
v_get_pixel( handle, x, y, pel, index )
WORD handle;
WORD x, y;
WORD *pel, *index;
```

In palette-based video modes, *pel* is the hardware-based value of the pixel and *index* is the GEM VDI pen value for the pixel. In true color video modes, the original GEM pen used to draw a pixel cannot be determined with any certainty, and the pixel value represents an RGB value, so the meanings of the values returned in *pel* and *index* change.

In 16-bit true color modes:

```
index = 0
pel = 16-bit RGB pixel value, where:

      Bit 15          Bit 0
pel:  (RRRR RGGG GGGB BBBB)

      Red = 0-31
      Green = 0-63
      Blue = 0-31
```

In 32-bit true color modes:

```
index = high word of 32-bit RGB pixel value
pel = low word of 32-bit RGB pixel value

       Bit 15          Bit 0
index: (AAAA AAAA RRRR RRRR)
pel:   (GGGG GGGG BBBB BBBB)


       A = Alpha Channel (non-RGB information)
       R = Red (0-255)
       G = Green (0-255)
       B = Blue (0-255)
```

vsf_udpat()

```
vsf_udpat( handle, pfill_pat, planes )
WORD handle;
long *pfill_pat;
WORD planes;
```

In palette-based modes, *pfill_pat* is a pointer to a 16x16 raster form in device-specific format. The raster may be either monochrome or color. In true color modes, for color fill patterns, the *pfill_pat* parameter should be a pointer to 256 (16 rows of 16 pixels each) 32-bit values which contain 24 bits of RGB information for each pixel of the fill pattern (using the format shown below). The *planes* value should be set to 32. This is true even for true color video modes with less than 24 bits per pixel. (GEM VDI will translate the RGB information to the correct values automatically.)

```
RGB format for user defined fill pattern data:

Bit 32                                 Bit 0
(0000 0000 RRRR RRRR GGGG GGGG BBBB BBBB)
```

Single-plane user-defined fill patterns work the same way in true color modes as in palette-based modes.

v_contourfill()

```
v_contourfill( handle, x, y, index )
WORD handle;
WORD x, y;
WORD index;
```

In true color video modes, the results of ***v_contourfill()*** may be different from palette-based video modes, although the function is still called in the same way.

In palette-based video modes, if the *index* parameter is a valid pen value, then the fill expands outward from *(x,y)* until it reaches pixels drawn with that pen. If *index* is negative, then the fill expands outward from (x,y) until it reaches pixels drawn with a different pen from the pixel at *(x,y)*.

For example, let's say VDI pen 12 and VDI pen 13 have both been set to RGB values of [1000,0,0] (pure red) with ***vs_color()***. If you have a big circle drawn with pen 12, and a smaller circle drawn with pen 13 inside that, and you fill at the center point of both circles with *index* equal to 12, then the fill will expand outside of the little circle until it reaches the big circle. It doesn't matter that both circle are shown in the same RGB color on-screen, because the VDI is able to determine that they have been drawn with different pens, because the values in the screen raster memory represent the pen values, not the RGB values.

In true color video modes, VDI will look up the current RGB values used for the pen indicated by *index* and the fill will stop when it reaches any pixel with the same RGB values. It doesn't matter what pen was used to draw the pixel; if a pixel's RGB values matches the current RGB value for the *index* pen, the fill will stop at that pixel. In the above example, that means the fill will stop when it reaches the smaller inside circle.

If *index* is negative in true color video modes, then the fill will expand outward from *(x,y)* until it finds pixels with a different RGB value from the pixel at *(x,y)*. It doesn't matter if the pixels were drawn with different pens, if they have the same RGB value as the pixel at *(x,y)* they will be filled.

SpeedoGDOS
(Continued from Page 2)

One result of this change is that there will now be a portion along each edge of the raster area that corresponds to the unprintable area of the printer, so anything your program outputs to the printer driver that falls within these areas will not be printed.

Since even "compatible" printers are slightly different from one to the next, the unprintable area can be adjusted in each driver, so you can fine-tune it for your individual printer. Also, you can turn the feature off altogether for compatibility with older programs which may have trouble printing with it turned on.

Portfolio Q&A

J. Patton

Menu Limitations

The internal menu routines (Int 60H Fn 1) are used extensively by the internal applications as well as many third party programs, but we have recently discovered a problem that the designers had probably never anticipated.

When using a list of over 250 entries, jumping backwards through a list (as is done when selecting the first letter of an entry does in the internal appications) of over 250 items the list will not be redrawn correctly until the default number of menu entries has been scrolled through. This is a fairly unique situation and disrupting to the user interface, not fatal. This probably stems from a combination of the OS being compiled in SMALL memory model and the intended use to only hold a few menu entries. In any event you can create your own menus to circumvent any restriction.

PowerBASIC

Q: I'm writing a program in PowerBASIC and after the program has run fine once I get strange errors. Sometimes there is garbage onscreen and values in the data fields, but I can't see anything wrong with my program.

A: It turns out that initializing your numeric variables is very important in PowerBASIC and not doing so caused the mysterious error you saw.

Q: On the PC I set the caps lock by doing the following:

```
DEF SEG = &H40
POKE &H17,&H40 or PEEK (&H17)
```

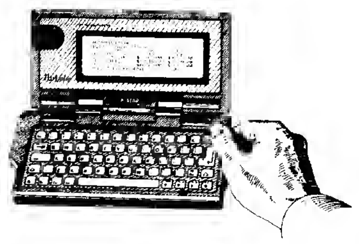
This doesn't work under PowerBASIC on the Portfolio.

A: One of the concessions which PowerBASIC made in order to fit on the Portfolio was the ability to understand hexadecimal notation. If you use decimal notation the code fragment above works as it did previously.

Battery Low Address

Q: I really need to have my program test for low battery state. How can I do this?

A: At location 8051h you can test for a value of less than 40 for a low battery.



Big Cards Available JEIDA card adaptors

There are now two sources of RAM/OTP cards greater than 128K bytes. These cards come in various sizes up to 2MB. DOS an applications less than 128k in size can take advantage of these cards transparently. Applications greater than 128k must keep track of which 128k page they are on. See the technical reference manual section 2.3.4.

Additionally Becker and Partner have a PCMCIA/JEIDA card adapter for the Portfolio which fits into the RAM card slot of the Portfolio. This adds about 1 1/2" to the length of the Portfolio.

Becker & Partner GmbH
Postfach 190
Wilhelmstraae 91
5100 Aachen
Germany
Tel: 49 241-50 90 18
49 241-50 90 19
Fax: 49 241-50 95 77

Microcard Electronics
La Faye 42220 Burdignes
France
Tel: 33 77 39 68 13
Fax: 33 77 39 19 60

Flash Memory

Flash Memory cards are now available from Optrol Inc. in sizes of 1MB, 2MB, and 4MB. Flash RAM requires no battery power and can rewrite up to 100,000 times.

Optrol Inc.
P.O. Box 37157
Raleigh, NC 27627
(919) 779 3377

Universal I/O Interface

BSE announces a Universal I/O Interface which is a combination serial/parallel peripheral with a 128k EPROM as brive B full of utilities. They have also announced a 512k RAM upgrade interface to give your Portfolio a full 640k of RAM.

The BSE Company
Tel: (602) 527 8843
Fax: (602) 527 1540

New Online

PDEMO.ZIP - Ultimate Portfolio Demo. Minimum requirement 286PC, 640k, VGA. In store demo as well as a source of information for those curious about the Portfolio.

PSND1.ZIP - Plays digitized sounds on the Portfolio.

MAKSND.ZIP - Creates sound samples to be played with PSND1.

(continued on page 20)

Atari MultiTOS User Interface Guidelines

Bill Rehbock

Application Elements

User-friendly GEM applications should provide the user with a consistent, predictable means of interacting with the computer. The most popular applications to-date have always been those that the user feels at home with, because of general familiarity with other applications that they have previously used. User interface design is a critical consideration during product development and should be well thought out before actually sitting down and laying out and coding the interface.

The basic elements of a GEM application are the menu bar, the application's window (or windows), dialog boxes, alert boxes, and if the application warrants them, toolbox windows. GEM applications may optionally install their own desktop background, which is swapped out by the AES to reflect the active application.

The Menu Bar

Applications should normally consist of a MENU BAR, which will generally have the titles from left to right, "***Prgname***", "***File***", "***Edit***", and then the additional application-specific main menu titles. "Prgname" should be replaced with the application name so that users can quickly identify which application's menu bar they are looking at.

For user convenience, the standard entries under "File" should start with "*New*", "*Open...*", followed by other load-oriented operations, then in the next section of the menu, "*Close*", "*Save*", "*Save as...*", and the other application-specific save-oriented functions. The next section down should be used for other file operations such as "*Import...*" and "*Export...*". This should be followed by the menu items for printing, usually "*Page Setup...*", then "*Print...*". The last item under "File" should always be "*Quit*".

Note: *A menu item must be followed by an ellipsis to indicate that additional action or input will be required by the user to carry out the requested task. For instance, "Save" indicates that the file will be saved directly, using the current name, whereas "Save as..." will require the additional input of a filename.*

The "***Edit***" menu should start with "Undo", then in the next section, "*Cut*", "*Copy*", "*Paste*", and "*Delete*". The rest of the "Edit" menu is usually application-specific, but the next menu item, if used should be "*Select all*".

If applicable, the fourth main menu title should be "***Options***", where menu items such as "*Document defaults...*", or "*Preferences...*" should appear.

Note: Menu titles and items should never be displayed in all uppercase letters. Menu titles should have one space before and after each title. There should be two spaces to the left of menu items.

Keyboard Equivalents For Menu Items

The standard sytem-wide keyboard equivalents that should be used system-wide for no other purpose other than those listed are:

[Control-N]	New
[Control-O]	Open
[Control-W]	Close
[Control-S]	Save
[Control-P]	Print
[Control-Q]	Quit
[Control-X]	Cut
[Control-C]	Copy
[Control-V]	Paste
[Control-A]	Select all
[Control-F]	Find
[Control-H]	Replace
[Control-G]	Find next
[Delete]	Delete
[Undo]	Undo
[Help]	Invoke help

Note: *The [Alternate] key is used as a character modifier on non-U.S. keyboards to access the necessary extended characters in applicable countries, and should not be used for keyboard equivalents in most cases.*

Windows

The primary stage for user interaction with the application is the window. Most of the user input, whether typing, drawing, or editing, is performed in the confines of windows. All of an application's output should be constrained to the application's own windows only. See the VDI and AES manuals for further information regarding window work areas and clipping rectangles.

Document windows should have, at a minimum, a mover/title bar so that even if the window is not resizable, the user can move the window off to the side of the desktop to have access to other items. The other window elements are the Info bar, Closer, Sizer, Full box, Sliders, and Arrows. The general use of these is apparent in the GEM Desktop. It should be noted that GEM sliders are always proportional so that the user has constant feedback as to the percentage of the document that is being viewed.

Operating system calls allow every element of windows to be set to any color and fill pattern. The user generally selects these attributes using the Window Colors CPX in the Control Panel and they should not be altered by an application. In video modes with greater than 16 colors, other than True Color, the first 16 color entries should be reserved for use by the system for drawing elements for which the user has set preferences.

Keyboard Equivalents for Cursor Movement Inside Windows

The system-wide standard for keyboard cursor manipulation is as follows:

[Control-Left/Right Arrow]	Move cursor to beginning of word to the left/right
[Control-Backspace]	Delete from cursor position to start of next word to the left
[Control-Delete]	Delete from cursor position to start of next word to the right
[ClrHome]	Move cursor to beginning of document
[Shift-ClrHome]	Move cursor to end of document
[Shift-Delete]	Delete line

Dialog Boxes

Dialog boxes are used for modal input. That is, input that the user must provide before any further processing may be done. They are generally used for parameter setting and other selections that require the undivided attention of the user. They should never be used for on-going informational or status output, as it would interfere with the normal real-time user interaction with the system.

Alerts

Alerts should be used to call the user's attention to conditions that develop that require immediate user knowledge. The simplest and most common would be an alert notifying the user that he is quitting an application without having saved the open document. Alerts should also be used to notify the user that a time-consuming or unalterable function is about to be performed.

Alerts usually have two or three buttons that allow the user to make some sort of decision based on the information provided. Alerts with only one button are very frustrating to the user, as it implies a lack of control over what is about to happen. The general rule for alerts is to have the "OK" button to the left of the "Cancel" button. "Cancel" should always be capitalized, and "OK" is uppercase.

Note: *Buttons in general should be capitalized words, not all uppercase.*

Toolbox Windows

Toolbox Windows are a special class of window that are used for providing the user with non-modal control or information. The most common use would be for drawing tool selection in a paint program, or color selection. The tools are usually shown as logical groups of icons that the user can easily associate with their functions. Another use of this type of window is continual status output, such as the progress of a file download or recalculation time.

Other General Notes

Applications should make no assumptions on what type of system the user will have. Be able to deal with any screen size and color resolution. Use the operating system calls to determine the screen dimensions and system capabilities to provide the user with the richest computing experience possible. Users have grown to expect unsurpassed ease of use from applications available for Atari computers. If you have any questions regarding user interface design for Atari computers, please feel free to call your developer support representative.

Game/Entertainment Software Guidelines

The following points should be followed:

* Installable on a hard disk.
* Should be able to be launched from any video resolution.
* The user should be presented with a single executable file; leave ancillary data files, high score files, etc. inside a companion folder.
* Allow the user to exit and return to the desktop exactly where and how they left off.
* Use the Falcon enhanced joystick for all joystick-oriented games; CX-40 style controls should not be supported.
* Ideally, where possible, allow the game to be run in a window; this is well-suited for users that want to play games in the MultiTOS multi-tasking environment (such as while downloading a file).
* We expect most users to run in 640x480x256 color mode; you may want to keep this in mind.
* If you store your screen data in VDI standard format, you can use the GEM VDI function vr_trnfm() (transform form) at runtime to convert the data into the correct format for the current video mode. This will allow you to use the same data on interleaved bitplane screens (like the built-in video modes) or a pixel-packed screen (like with some add-on video cards).

Zen and the Art of Metafile Maintenence

Mike Fulton

GEM Metafiles are a special type of vector graphics file that consist of a series of GEM VDI functions that are used to create a picture. Unlike other vector graphics formats like CVG (Calamus Vector Graphic) or EPS (Encapsulated Postscript), the information in a GEM metafile is 95% ready to be passed straight to GEM VDI. The only processing that is normally required to display a metafile is to scale and translate the coordinates for each graphics call so that the image appears at the correct size on the correct part of your screen or other device.

If you already know how to output to other GEM devices such as the display screen and the printer, then you already know most of what you need to start creating GEM Metafiles. But there are a few additional features that go along with metafiles which don't apply to other devices.

First of all, with other GEM devices, when you open the workstation you get back information which indicates the size of the display area in pixels, along with some other information that tells you the size of a pixel. You can then use these different pieces of information to calculate the size of the display area in real-world measurements like inches or millimeters. Once you've done this, it's not too difficult to scale and position your output so that things come out in the right place regardless of the resolution of the device involved.

This is also true for metafiles, but there are additional GEM VDI functions which allow your program to customize the metafile 'world' to suit your needs.

Page Size

First of all, with GEM metafiles your program can specify a particular page size, rather than just taking what you get. The ***vm_pagesize()*** function is used for this purpose:

```
void vm_pagesize( handle, width, height )
WORD handle, width, height;
```

The *width* and *height* parameters specify the size of the page in tenths of a millimeter. So for an 8" x 10" page, you would use values of 2032 for the width and 2540 for the height. What you actually specify as your page size is less important than the requirement that you must set it to something. At the very least, specify some default page size like 8" x 10" or 10" x 8" (the page size should be proportional to the format of the drawing). (Note that since 16-bit values are used for the page size, there is a limit of approximately 129 inches on both the width and height. This will probably not be a problem for most programs.)

If you create a metafile without specifying the page size, then other programs may not be able to correctly scale the objects in your metafile for display, and some may not accept the metafile at all.

This function changes the width and height fields contained in the header of the GEM metafile being created so that when another program reads the metafile, it will be able to determine the page size so that it can draw objects from the metafile in the proper size and proportions.

World Coordinates

With all GEM devices other than metafiles, the coordinate system is similar to the to top-right quadrant of the basic Cartesian system, except that the y-axis is reversed (as shown in figure #1).

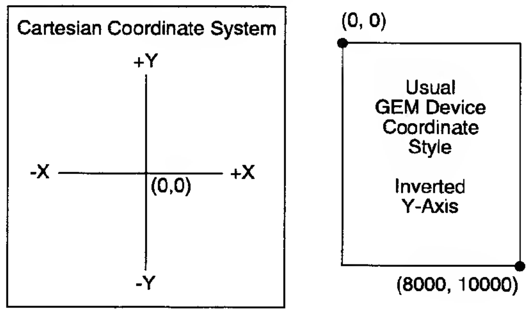


Figure #1

While the default coordinate system is the same as other GEM devices, metafiles also allow the use of different coordinate systems. The ***vm_coords()*** function allows your program to specify the coordinate range used in the metafile 'world'.

```
void vm_coords( handle, min_x, min_y, max_x, max_y )
WORD handle, min_x, min_y, max_x, max_y;
```

This function changes the coordinate system used by the GEM metafile currently being created. The *min_x* and *min_y* parameters should contain the coordinates for the top left corner of the page and the *max_x* and *max_y* parameters should contain the coordinates for the bottom right corner of the page. The parameters you give ***vm_coords()*** are placed into the header of the metafile being created.

You can set up nearly any style of coordinates you desire with this function. If the usual GEM coordinate style doesn't suit your needs, you can use a different style. For example, if you prefer a true Cartesian coordinate system, then you can set up one, with the (0,0) coordinate in the center of the display area, as shown in figure #2.

Note: *The **vm_coords()** and **vm_pagesize()** functions only affect the header of the metafile being created. They do not affect information returned from GEM VDI's various inquire functions.*

For example, the current version of the metafile driver sets up a default pixel size of 85 microns width and height, or 300 dots per inch. If your program were to

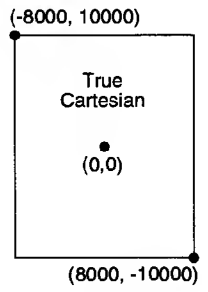


Figure #2

*set a page size of 8" x 10" with the **vm_pagesize()** function and a coordinate range of (0,0) to (8000,10000) with the **vm_coords()** function, then that works out to 1000 dots per inch in the metafile 'world' you have set up. However, the values returned by the **vq_extend()** function which specify pixel width and height or device resolution will not change from their original values.*

*Furthermore, if your program uses font metrics information obtained from the metafile driver, it will still be based around the originally specified pixel size. For example, if you set the text size to 72 points, then do a **vqt_fontinfo()**, **vqt_width()**, **vqt_extent()**, or any other function which returns information about text size, you will get back a value that represents the original resolution, not 1000 dots per inch.*

In most cases, this will probably not be a problem, as your program will not be positioning objects (including text) using information obtained from the metafile driver, but rather using information obtained from the screen or printer driver.

Metafile Filename

By default, the metafile driver creates a file named GEMFILE.GEM in the current default directory. However, the metafile driver allows you to specify the filename of the metafile to be created.

```
void vm_pagesize( handle, fname )
WORD handle;
char *fname;
```

The *fname* parameter should be a pointer to a string containing a valid GEMDOS filename specification.

In order for it to be successful, the ***vm_filename()*** function must be called immediately after opening a metafile workstation. The metafile driver then creates the specified file and writes out a metafile header.

Note: *The GEMFILE.GEM file created at open-workstation time is not deleted by the **vm_filename()** function. It is the application's responsibility to delete this file.*

Metafile Bounding Box

The page size and coordinate system work together to define the size and shape of the metafile 'world', but they say nothing about what inhabits that world. If the metafile header indicates a page size of 8" x 10" and a coordinate system of (0,0)-(8000,10000), how do you determine what portion of the page actually contains objects?

```
void v_meta_extents( handle, min_x, min_y, max_x, max_y )
WORD handle, min_x, min_y, max_x, max_y;
```

This function is used to specify the area of the page that actually contains objects. The *min_x, min_y, max_x,* & *max_y* parameters specify a bounding box that can contain all objects in the metafile. This allows an application reading a metafile to easily determine, for example, that although the page size is 8" x 10", the objects on the page only occupy a 2" x 2" square at the top left corner. This allows the application to ignore the rest of the page and display just the area that has stuff in it.

Note: *There is nothing to prevent the bounding box from extending outside of the metafile's specified coordinate range. Objects may extend outside the boundaries of the page, provided that they can exist within the limitations of the coordinate system used for the metafile. (As seen in figure #3)*

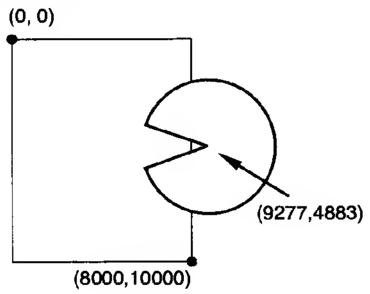


Figure #3

However, if you had instead a coordinate range of (0,0) at the top left corner to (32767,32767) at the bottom right corner, then you couldn't extend an object past the right side or bottom of the page because some of the coordinate values would become negative (GEM VDI uses signed 16-bit values). This would distort the appearance and/or location of the object. (See figure #4.) This is another reason to be careful in deciding what sort of coordinate range to use.

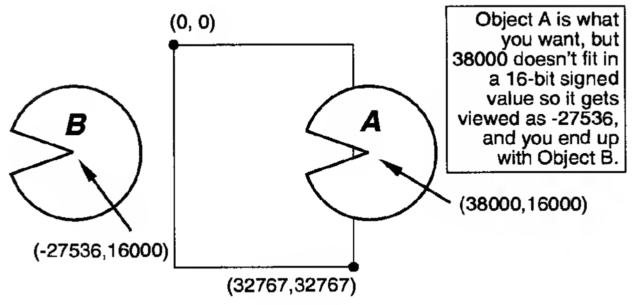

(0, 0)
Object A is what you want, but 38000 doesn't fit in a 16-bit signed value so it gets viewed as -27536, and you end up with Object B.
B
A
(38000,16000)
(-27536,16000)
(32767,32767)

Figure #4

For display purposes, if a metafile's bounding box does extend outside the page, an application should display only that portion which falls within the page.

To keep track of your metafile bounding box, simply keep a set of variables for minimum and maximum coordinates used, and prior to each output function, check the coordinates and update your variables if necessary. Prior to closing the metafile, do the *v_meta_extents()* call. This involves a little extra work that isn't necessary for other GEM devices, but it's the most straightforward method.

At the minimum, you should call *v_meta_extents()* with the same parameters you passed to *vm_coords()* so that the metafile header will contain something besides zero for the bounding box values.

Metafile Escape Functions

Normally when you do a GEM VDI call such as *v_bar()* using a metafile workstation handle, the metafile device driver writes out the contents of the GEM VDI *CONTRL[]* array, which includes the opcode that indicates a *v_bar()* function, as well as how many values are contained in the *INTIN[]* and *PTSIN[]* arrays. Next it writes out the specified number of words from the *INTIN[]* array, and then the specified number of words from the *PTSIN[]* array. This results in all the information for the *v_bar()* function being placed into the GEM metafile.

However, when saving a document, many applications want to include additional information for their own use. The metafile escape functions provide a method to do just that. They allow a program to save information into a metafile which isn't directly displayable by GEM VDI.

```
void
v_write_meta( handle, intin_len, intin, ptsin_len, ptsin )
WORD handle;
WORD intin_len, *intin;
WORD ptsin_len, *ptsin;
```

As with other GEM commands, the contents of the *CONTRL[]* array will be written to the metafile, indicating a metafile escape function. The *intin* parameter is a pointer to an array of WORD values that will be written to the metafile. The *intin_len* parameter indicates the length of this array. The *ptsin* parameter is a pointer to an array of WORD values that will be written to the metafile. The *ptsin_len* parameter indicates the length of this array.

The *intin[0]* value is used as a sub-opcode to indicate the purpose of the information. This can be something specific to your application, or it can be one of several pre-defined sub-opcode values from the list below:

10	***Start Group***
11	***End Group***
49	***Set No Line Style***
50	***Set Attribute Shadow On***
51	***Set Attribute Shadow Off***
80	***Start Draw Area Type Primitive***
81	***End Draw Area Type Primitive***

Other sub-opcodes in the range of 0-100 are reserved. Application-specific sub-opcodes can be anything from 101-65535. The main sub-opcodes we're concerned with are ***Start Group*** and ***End Group***. For more detailed descriptions of all the pre-defined sub-opcodes, please see Appendix H of your GEM VDI manual.

Let's say your application is a drawing program, and the user has created a drawing with 20 objects in it. Ten of these objects are grouped together, so that from within the program, when you move or size or rotate one of them, you do it to all the other objects within the group as well. When you save the file, you want to include some information that will specify that these objects are grouped together, so that the next time the drawing program loads the picture, these objects will still be treated as a group.

By putting the ***Start Group*** sub-opcode into *intin[0]* and using the v_write_meta() function, a special metafile escape function will be placed into the metafile. After the last object of the group has been written, then the application would place the ***End Group*** sub-opcode into *intin[0]* and do another *v_write_meta()* call.

Later when your application reads the metafile back, it will get to the metafile escape function and find the ***Start Group*** sub-opcode, and know that the objects coming afterwards are part of a group. When it gets another escape function with an ***End Group*** sub-opcode, then the group ends. (Note that a group may contain other smaller groups.)

Application-Specific Metafile Escape Functions

What other useful information might an application want to put into a file? Wouldn't be nice if the application could determine the bounding box of a group without looking at each object in the group? You could use an application-specific sub-opcode that your application understands to mean "***Group Bounding Box***", and immediately after the application writes out the ***Start Group*** code it would then do a *v_write_meta()* function that writes out the "Group Bounding Box" sub-opcode along with the coordinates for a bounding box for the entire group.

Suppose your program had taken some text and grabbed the character outlines from FSMGDOS, and was going to place the outlines into the metafile as bezier curve objects. Let's say the original text string reads "Atari". The problem is, if you don't put anything but the bezier curve objects into the metafile, then you don't have any way to get back to the "Atari" string and change it, except by having the user delete the "Atari" bezier objects and then repeat the whole process for a new string.

Your application could start a group, and then do a *v_write_meta()* funtion with a special application-specific sub-opcode that means "***Group Contains Text Outlines. Original Text and Font ID:***" along with the original "Atari" string and font ID (or fontname string) for the original text. This would be followed by bezier objects for the text.

With a program that understood what the special sub-opcode meant, it would be easy to go back to the original "Atari" string, edit it, and then get a new set of

outlines, with minimal effort on the user's part.
If the metafile was being displayed by something that didn't understand the special sub-opcode, it would just see the bezier curve objects and draw them, ignoring the escape function. The picture would end up looking correct, even though this program didn't understand the entire file and wouldn't be able to edit it in as flexible a manner.

Suppose you wanted to bind some text to a bezier curve path. The metafile could include the path curve in an escape function (so that it doesn't get displayed by other programs, since it's just a guide for the text). Then it would write out bezier curve objects taken from the text outlines after they've been warped around the text path bezier curve.

A program that understood the special sub-opcode would know that it should allow the user to edit the bezier curve contained in the escape function, and that if the user changed that first bezier curve, it should reposition the text outlines along the new path. And this could be combined with the previous example so that you could even change the text string to get a completely new set of character outlines. A program that didn't understand would just draw the bezier objects for the text outlines, and the picture would look correct.

The possibilities are endless.

Since the sub-opcodes for application-specific metafile escape functions for one program could conflict with another program, I would urge two things:

First, put a special magic number into *intin[1]*. If your program reads a metafile escape function and the magic number in *intin[1]* doesn't match the sub-opcode in *intin[0]*, then ignore the escape function.

For example, let's say our ***Group Bounding Box*** sub-opcode is 3000. The magic number for this sub-opcode will be 1701. If we read a sub-opcode of 3000, then we'll check *intin[1]* for a value of 1701 and ignore the escape function if we find any other number there.

Secondly, when you create an application sub-opcode, prior to the program's release, please inform Atari Developer Support at Atari Corporate HQ of what the sub-opcode is, and what it's supposed to mean. This way if there are any conflicts with anything else, they can be fixed before your program is released.

Reading & Displaying GEM Metafiles

Displaying metafiles is for the most part even easier than creating them. But when reading a metafile there is one question that needs to be answered before you do anything else: Is the metafile being read going to be displayed only, or will the user have the capability of editing it?

If the metafile is being read for display only, then your task is made much simpler. GEM metafiles require only a minimal amount of processing in order to be displayed at virtually any size on any GEM device.
There is a sample program which accompanies this article that reads and displays GEM metafiles inside a window. Unfortunately, it's a little too long for the entire program listing to be included here, so only the listings for the metafile display routines are shown. You can find the METAFILE.ARC file, containing the sample program and all of the source code, in the Atari developer areas of the GEnie (Library of ATARI.RSC Roundtable) and Compuserve (Data Library 7 of ATARIPRO forum) on-line services. If you don't use these online-services, you may contact Atari Developer Support and request the program.

The source code is setup to work with either Mark Williams C, Lattice C v5, or Alcyon C. It should be easy to get it to work with other compilers by changing the #include statements at the top of the METASHOW.C file.

The sample program looks for a GEM metafile named TEST.GEM at startup-time, and opens a window and displays the picture inside it. You may resize the window and the image will be resized (it will keep the proper proportions, however). You may also view information from the metafile header and view other GEM metafiles by using the choices in the drop-down menus.

The ***display_metafile()*** function in the METASHOW.C (listing not included here) file is the window-redraw function. It determines the proper width and height required to show the metafile at the largest possible size within the window without changing the original aspect ratio. Then it calls the metafile display routine.

The ***meta_show()*** function in the SHOWMETA.C source code file (listing #2) is the actual metafile display routine. It is not small, but it is fairly straightforward. It reads the metafile header to determine the page size, aspect ratio, and coordinate system used, and then it calculates some scaling factors using this information. After this, it enters the main display loop. Here it reads information from the metafile that represents the GEM VDI *contrl[]*, *intin[]*, and *ptsin[]* arrays. This represents all of the information we need to pass along to GEM in order to call a GEM VDI function.

However, there is one last step required before we can pass the information along to GEM VDI. The coordinates for all of the objects in the metafile have been positioned and sized according to the page size and coordinate system used by the metafile. Before we can draw them on our output device, object coordinates need to be need to be scaled and translated to the proper size and position. In the case of the sample program, that is the size we determined earlier: the work area of the program window.

The display routine uses a few external support routines for scaling coordinate values. There is also a routine that gets 16-bit values out of the metafile buffer (making it easy to modify things so that the metafile doesn't have to be completely in memory to be displayed).

The display routine is built around the idea that most metafile items are either ready to be passed directly to

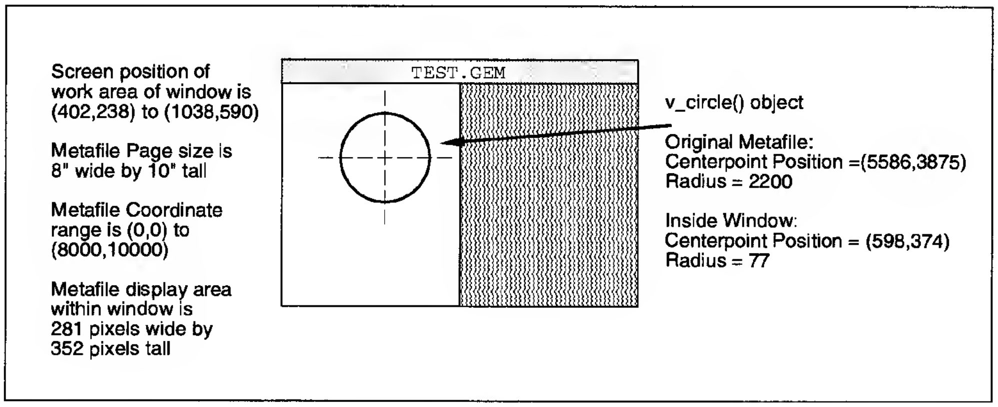

Screen position of work area of window is (402,238) to (1038,590)
Metafile Page size is 8" wide by 10" tall
Metafile Coordinate range is (0,0) to (8000,10000)
Metafile display area within window is 281 pixels wide by 352 pixels tall
TEST.GEM
v_circle() object
Original Metafile: Centerpoint Position =(5586,3875) Radius = 2200
Inside Window: Centerpoint Position = (598,374) Radius = 77

Figure #5

GEM VDI, such as attribute setting functions like ***vsf_style()*** or ***vsl_color()***, or else they just need to have a set of coordinates scaled and translated, like ***v_pline()*** or ***v_bar()***. These functions can all be handled together in a standard way, so you are left with the task of catching all of the known special cases. These include GEM VDI functions which have size values of some kind in *ptsin[]*, rather than coordinates, or which have size values in the *intin[]* array. Examples would include ***vsl_width()*** where line width is specified in *ptsin[0]*; ***vst_height()*** where text height is specified in *ptsin[1]*; and ***v_ellipse()*** where the *ptsin[]* array includes coordinates as well as a set of radius values for the x-axis and y-axis.

These special cases are handled by figuring out which *ptsin[]* or *intin[]* values are special, and then using a different scaling function for these values which scales the appropriate value and doesn't translate it like it would with a coordinate value.

In figure 5, we see a metafile with a ***v_circle()*** object displayed in a GEM window. This object provides an example of a special case object that requires both coordinate and size-value scaling. The display routine sees that the object is a GDP-circle object, so it starts out by scaling and translating the centerpoint coordinates in *ptsin[0]* & *ptsin[1]*. This is done by calling the ***xoffset_scale()*** and ***yoffset_scale()*** functions.

Translating the coordinates is fairly simple. We take the ratio of the display area coordinate range (281x352) to the metafile's coordinate range (8000x10000) and multiply the coordinates by this value (each axis is done separately). Now the circle's coordinates have gone from (5586,3875) to (196,136). Now we add the window position so that the screen coordinates of the circle end up being (598,374). The scaling routine used by the metafile display function handles both the scaling and translation in one step. The ***meta_show()*** function sets up offset values that the scaling routines use to account for whatever the metafile's coordinate system might be, say like that shown in figure 2.

For the radius value, the step of translating the coordinates into the window is not required, so that value is scaled by a different routine which skips that step.

For GEM VDI functions where the *ptsin[]* array is empty, or where it contains coordinates only, such as ***v_pline()***, ***v_bar()***, or ***vs_color()***, the display routine sees that they are not a special case function. If the *ptsin[]* array contains coordinates, then it calls functions that scale them to the correct size and translates them to the desired position on the output device. Then it lets the call fall through to GEM VDI.

Note: *For any function not recognized as a special case, the display routine will pass it along to VDI on the assumption that VDI will know what to do with it. Any values in the ptsin[] array are scaled and translated as coordinates. Values in the intin[] array are left unchanged.*

If a metafile contains a completely brand new function, the display routine should still work on any machine that supports the new function. This is because it passes everything through to GEM VDI. If GEM VDI doesn't recognize the function, it will be ignored. At the very worst, the metafile might contain some objects that are displayed incorrectly, but this is better than not displaying anything.

Another example of a special case might be where a particular function is not supported on a certain device (like ***v_bit_image()*** for the screen), and you want to handle it by drawing it with your own function instead of passing the call to GEM VDI. The ***meta_show()*** function doesn't do anything for ***v_bit_image()*** on the screen, but there are some comments and a disabled section of code that makes plugging in such support a simple matter.

The ***meta_show()*** function does not look at the bounding box information of the metafile. If an application wants to show just what's in the bounding box area, it should obtain the bounding box information using the ***meta_info()*** function, and then adjust the world coordinates and size to zoom in on that particular section, and then adjust the clipping rectangle accordingly. There are several reasons why it's done this way. First, some programs which write out metafiles either don't write out the bounding box information correctly or don't write it at all. Also, it takes a level of control away from the program calling ***meta_show()*** since it could then not zoom out away from the bounding box. You may want to see the entire metafile 'page', not just the part that has something on it.

Reading and Editing GEM Metafiles

If your application wants to edit the contents of GEM metafiles once they've been read in, then you have a choice of either maintaining the original data structure or translating it into your own data structure.

In the former case, if the metafile contains escape functions your application doesn't understand, it should not remove them until the user does something that requires changes to the metafile. That is, if the user loads a metafile, just looks at it without changing anything, then saves it back out again as a metafile but maybe with a new filename, then the new file your application creates should be identical to the file that was read in the first place. You'd expect a text editor to work this way, so why shouldn't a graphics editor?

Once the user does anything that requires changes to the metafile, then any escape functions your program doesn't understand should be removed, because the changes your application makes to the metafile may make the escape functions invalid. From that point, the application may make whatever changes and add any escape functions of its own that are appropriate.

Some applications will want to work with other file formats other then GEM metafiles. For example, they may want to read in a GEM metafile and write out an EPS (Encapulated Postscript) file. In these cases, the application will most likely want to translate the contents of an imported metafile into its own internal format.

The METAFILE.H file (listing #1) contains definitions of the GEM VDI opcodes for most of the functions you'll find in a metafile (if anything is left out, see your GEM VDI manual).

With a few changes, the ***meta_show()*** function in listing #2 could serve as the skeleton for a metafile-to-internal format translation function. Basically you'd want to add code to handle the non-special case functions instead of just letting them fall through. When you fall through to the ***vdi()*** call at the bottom of the main loop, you'd save a new object with all of the current attributes.

Listing #1

```
/**************/
/* METAFILE.H */
/**************/

/********************************/
/* These should be 16-bit values */
/********************************/

#ifndef UWORD
#define UWORD unsigned short
#endif

#ifndef WORD
#define WORD short
#endif
```

```
/****************************************************************************/
/****************** GEM VDI Drawing Command Opcodes ***********************/
/****************************************************************************/

#define V_CLOSEWORK      2
#define V_CLOSEVWORK     101
#define V_CLEAR          3
#define V_UPDATE         4
#define ESCAPE           5
#define PLINE            6
#define PMARKER          7
#define V_GTEXT          8
#define FILLAREA         9
#define GDP              11
#define VST_HEIGHT       12
#define TEXT_ROTATION    13
#define SETCOLOR         14
#define LINE_TYPE        15
#define LINE_WIDTH       16
#define LINE_COLOR       17
#define MARKER_TYPE      18
#define MARKER_HEIGHT    19
#define MARKER_COLOR     20
#define VST_FONT         21
#define VST_COLOR        22
#define FILL_INTERIOR    23
#define FILL_STYLE       24
#define FILL_COLOR       25
#define WRITE_MODE       32
#define VST_ALIGNMENT    39
#define CONTOURFILL      103
#define FILL_PERIMETER   104
#define VST_EFFECTS      106
#define VST_POINT        107
#define LINE_ENDSTYLE    108
#define SET_USERFILL     112
#define SET_USERLINE     113
#define RECTFILL         114
#define SET_CLIP         129

#define V_FTEXT          241    /* New opcodes added for FSMGDOS support */
#define VST_ARBPOINT     246    /* Beziers not included since they */
#define VST_SETSIZE      252    /* share opcodes with v_pline & v_fillarea */
                                /* No checking for FSMGDOS is done! */

/****************************************************************************/
/****************** GEM GDP Graphics Primitives subopcodes *****************/
/****************************************************************************/

#define BAR     1
#define ARC     2
#define PIE     3
#define CIRCLE  4
#define ELLIPSE 5
#define ELLARC  6
#define ELLPIE  7
#define RBOX    8
#define RFBOX   9
#define JUSTEXT 10

/****************************************************************************/
/************************ GEM VDI Escape subopcodes ************************/
/****************************************************************************/

#define BITIMAGE        23

/****************************************************************************/
/********************** GEM Metafile Header Structure **********************/
/****************************************************************************/

typedef struct metainf
{
    WORD    magic;      /* This stuff comes from metafile header */
    WORD    hdrlen;
    WORD    version;
    WORD    ndcflag;

    WORD    minx, miny;
    WORD    maxx, maxy;

    WORD    pg_wid, pg_ht;

    WORD    llx, lly;
    WORD    urx, ury;
    WORD    wc, hc;      /* this stuff gets calculated */
    long    length;
} Metafile_Info;
```

Listing #2

```
/*
 * SHOWMETA.C
 * GEM Metafile Display Routines
 * Written by Mike Fulton, Last Modified 1:35pm, 6/10/92
 *
*/

#define min(a,b) (a<b)?a:b
#define max(a,b) (a>b)?a:b
#define CLEAR_BACKGROUND 1

#include "metafile.h"

/****************************************************************************/

long world_x, world_y, world_w, world_h, area_wd, area_ht, x_offset, y_offset;

UWORD *bufpos;

/****************************************************************************/
/**************** Basic pre-ANSI style function prototypes. *****************/
/****************************************************************************/

void meta_show(), xoffset_scale(), yoffset_scale(), xscale_only(), yscale_only();

WORD get_wd(), intersect();

/****************************************************************************/

#ifdef LATTICE

#include "vdicall.h"

WORD contrl[20], intin[128], ptsin[256], intout[128], ptsout[256];
WORD *vdipb[] = { contrl, intin, ptsin, intout, ptsout };

#else

extern WORD intin[], ptsin[], contrl[];

#endif

/*************************************************************************/
/* meta_show() will display all GEM commands stored in a metafile buffer */
/* Last item in buffer should have opcode of 0xffff, as per GEM standard */
/* After this call, workstation attributes will most likely be          */
/* drastically altered!!! Do not depend on anything!                    */
/*                                                                       */
/* This function does NOT do the following:                              */
/*                                                                       */
/*    1) Load fonts                                                      */
/*                                                                       */
/*    2) Check for GDOS, FSMGDOS, FONTGDOS, anything else                */
/*                                                                       */
/*    3) Verify that buffer contains a metafile, not garbage             */
/*                                                                       */
/*    4) Display IMG files on screen devices when processing a           */
/*       v_bit_image() VDI call. (Unless using a VDI screen driver that */
/*       supports the call (the built-in screen drivers do not).         */
/*       See comments below if you want to add your own                  */
/*       screen device support for this function.                        */
/*                                                                       */
/*    5) Hide/Show mouse pointer                                         */
/*************************************************************************/
/* Input parameters:                                                     */
/*                                                                       */
/*         scrnhndl   - screen VDI workstation handle                    */
/*         dev_handle - Output Device VDI workstation handle             */
/*         dev_hsize  - Output Device pixel horizontal size              */
/*         dev_vsize  - Output Device pixel vertical size                */
/*                                                                       */
/*         metafile   - Pointer to start of metafile information         */
/*                      (buffer need not be word aligned)                */
/*                                                                       */
/*             wx, wy - Position of top left corner of area to draw the */
/*                      image. Specified in device coordinates.          */
/*                                                                       */
/*             ww, wh - Image box width and height. These values are    */
/*                      specified in device pixels.                      */
/*                                                                       */
/*             endbuf - pointer to end of metafile information           */
/*                                                                       */
/*            winrect - pointer to function used to calculate clipping  */
/*                                                                       */
/*************************************************************************/
/* You can show the metafile at various magnifications by changing the  */
/* (wx,wy,ww,wh) position and size. These coordinates can start outside */
/* the device raster by using negative coordinates, and be larger than  */
/* the device raster if required. However, the clipping parameters must */
/* be set to the portion of the output area that corresponds to the     */
/* actual device raster area (or output rectangle within the raster     */
/* area). Also see the notes on the winrect parameter below.            */
/*                                                                       */
/*************************************************************************/
/* There is an external subroutine which is supposed to insure that your */
/* output is confined to the proper portion of the screen or other      */
/* device. The show_meta() function expects your program to pass a      */
/* pointer to this subroutine function in the 'winrect' parameter.      */
/*                                                                       */
/* The winrect() function is passed an array of 16-bit signed values    */
/* containing the device coordinates of a clip rectangle calculated from */
/* the (wx,wy,ww,wh) world coordinates which was passed to show_meta(). */
/* The rectangle is a VDI-style rectangle, with coordinates of two      */
/* diagonally opposite points, not an AES-style rectangle with a        */
/* position and size.                                                    */
/*                                                                       */
/* The returned intersection rectangle should also be returned in the   */
/* input array. Then meta_show() will set the clip rectangle to the     */
/* returned rectangle.                                                   */
/*                                                                       */
/* If there is no intersection rectangle, the function returns '0',     */
/* telling meta_show() that it can quit before processing the metafile  */
/* information, because the output rectangle isn't visible.             */
/*                                                                       */
/* If you do not want to use such a function, then pass a NULL pointer  */
/* instead. Then the function will not be called, and the clipping      */
/* rectangle will be based on the output coordinates passed to the      */
/* meta_show() function through the (wx,wy,ww,wh) parameters.           */
/*                                                                       */
/* As implemented in the example below, winrect() calculates the        */
/* intersection of the passed rectangle and the work area of the window, */
/* then the intersection of that rectangle and the actual screen.       */
/*                                                                       */
/* For non-screen devices, you would want to output the rectangle of the */
/* desired metafile output area, intersected with the device raster     */
/* area. You might want to use a different function for different       */
/* devices or create a global flag that the winrect() function can check */
/* which you can set before calling the meta_show() function.           */
/*                                                                       */
/* The winrect() function is somewhat application-specific, which is why */
/* it is implemented outside this file and called through a function    */
/* pointer. An example of how to implement this function is shown       */
/* below. To adapt it to your application, simply substitute the proper */
/* variables for the window work area and screen resolution. Note that  */
/* the function intersect() is included in this file.                   */
/*                                                                       */
/*      int                                                              */
/*      winrect( r )                                                     */
/*      short *r;                                                        */
/*      {                                                                */
/*      short x, r2[4];                                                  */
/*                                                                       */
/*              r2[0] = theWindow.wx;                                    */
/*              r2[1] = theWindow.wy;                                    */
/*              r2[2] = r[0] + theWindow.ww - 1;                         */
/*              r2[3] = r[1] + theWindow.wh - 1;                         */
/*              x = intersect( r2, r );                                  */
/*              if( ! x )                                                */
/*                return( x );                                           */
/*              r2[0] = 0;                                               */
/*              r2[1] = 0;                                               */
/*              r2[2] = screen.xres;                                     */
/*              r2[3] = screen.yres;                                     */
/*              intersect( r2, r );                                      */
/*              return( x );                                             */
/*      }                                                                */
/*************************************************************************/
```

```
void
meta_show( scrnhndl, dev_handle, dev_hsize, dev_vsize,
           metafile, wx, wy, ww, wh, endbuf, winrect )

WORD scrnhndl, dev_handle, dev_hsize, dev_vsize,
     *metafile, wx, wy, ww, wh, *endbuf, (*winrect)();
{
WORD  points, header_length;
WORD  pagex, pagey, llx, lly, urx, ury, *tmppos;
WORD  clip[4], glclip[4];
register WORD opcode, dummy;
long wmicrons, h_ptsize_factor, v_ptsize_factor;

/*********************************************/
/* Start by reading image header information */
/*********************************************/

    bufpos = metafile;      /* Leave original pointer alone */
    tmppos = bufpos;        /* Save for later */

/*****************************/
/* Set up our global variables */
/*****************************/

    world_x = wx;
    world_y = wy;
    world_w = ww;
    world_h = wh;

/**********************************/
/* Read metafile header information */
/**********************************/

    dummy = get_wd();           /* $ffff for _normal_ metafiles */
    header_length = get_wd();   /* Length of header, in words */

    bufpos += 6;        /* Skip next 5 elements of header */

    pagex = get_wd();   /* Page width in mm/10 */
    pagey = get_wd();   /* Page height in mm/10 */

    llx = get_wd();     /* Lower left coordinate of page */
    lly = get_wd();     /* Lower left coordinate of page */
    urx = get_wd();     /* Upper right coordinate of page */
    ury = get_wd();     /* Upper right coordinate of page */

    bufpos = tmppos + header_length;   /* Skip past header */

/***************************************************************************/
/* If no metafile page size and coordinate range, then use some defaults!  */
/***************************************************************************/

    if( ! (pagex|pagey) )
    {
        pagex = 2032;   /* 254 microns per inch */
        pagey = 2032;   /* 8" square image */
    }

    if( ! (llx|lly|urx|ury) )   /* If no coordinate range specified */
    {
        llx = 0;                /* Then use default GEM-style coordinates */
        lly = 32767;
        urx = 32767;
        ury = 0;
    }

/*****************************************************************/
/* Set vars, Calculate size & offsets for metafile world & items */
/*****************************************************************/

    area_wd = urx - llx;        /* Used in scale & offset */
    area_ht = lly - ury;        /* functions */
    x_offset = 0;               /* Default offsets = 0 */
    y_offset = 0;

    if( llx != 0 )          /* Change offsets if necessary */
      x_offset = -(llx);

    if( ury != 0 )
      y_offset = -(ury);

/****************************************************************************/
/* Divide metafile page size by size of world to get pointsize scale factors */
/****************************************************************************/

    wmicrons = world_w * (long)dev_hsize;          /* width in microns */
    wmicrons = wmicrons * 10;                      /* Adjust for percentage */
    h_ptsize_factor = wmicrons / (long)pagex;      /* percent*1000 */

    wmicrons = world_h * (long)dev_vsize;          /* height in microns */
    wmicrons = wmicrons * 10;                      /* Adjust for percentage */
    v_ptsize_factor = wmicrons / (long)pagey;      /* percent*1000 */

/*****************************/
/* Set starting clip rectangle */
/*****************************/

    ptsin[0] = wx;                /* Clip rectangle */
    ptsin[1] = wy;
    ptsin[2] = wx + ww - 1;
    ptsin[3] = wy + wh - 1;

    if( winrect )
    {
        dummy = (*winrect)( ptsin );      /* Get external clipping */
        if( ! dummy )                     /* rectangle from the */
          return;                         /* specified function */
    }
    glclip[0] = clip[0] = ptsin[0];       /* Set clipping to */
    glclip[1] = clip[1] = ptsin[1];       /* where we're at now. */
    glclip[2] = clip[2] = ptsin[2];
    glclip[3] = clip[3] = ptsin[3];
    vs_clip( dev_handle, 1, ptsin );

/****************************************************/
/* Clear out the background of the slice.  This step */
/* may be considered optional for some purposes.     */
/****************************************************/

#if CLEAR_BACKGROUND
    vsf_interior( dev_handle, 2 );
    vsf_style( dev_handle, 8 );
    vsf_perimeter( dev_handle, 0 );
    vsf_color( dev_handle, 0 );

    ptsin[0] = world_x;
    ptsin[1] = world_y;
    ptsin[2] = world_x + world_w - 1;
    ptsin[3] = world_y + world_h - 1;
    v_bar( dev_handle, ptsin );
#endif

/********************************/
/* Bump up Bezier drawing quality */
/********************************/

    contrl[0] = 5;      /* Don't use a library binding */
    contrl[1] = 0;      /* because some compilers may not */
    contrl[2] = 0;      /* have a v_bez_qual() binding yet. */
    contrl[3] = 3;
    contrl[4] = 1;
    contrl[5] = 99;
    contrl[6] = dev_handle;
    intin[0] = 32;
    intin[1] = 1;
    intin[2] = 99;

#ifdef LATTICE
    vdi( vdipb );
#else
    vdi();
#endif


/******************************************/
/* Loop through metafile & playback commands */
/******************************************/

    do
    {
        contrl[0] = get_wd();          /* Read VDI opcode */
        contrl[1] = get_wd();          /* Read PTSIN count */
        contrl[3] = get_wd();          /* Read INTIN count */
        contrl[5] = get_wd();          /* Read sub-opcode */
        contrl[6] = dev_handle;        /* Set device handle */

        opcode = contrl[0];
        if(opcode == (short)0xFFFF)     /* Reached end of metafile? */
          break;

/* Read PTSIN & INTIN array values */

        for( dummy = 0; dummy < contrl[1]*2; dummy++ )
          ptsin[dummy] = get_wd();

        for( dummy = 0; dummy < contrl[3]; dummy++ )
          intin[dummy] = get_wd();

/**********************************************************/
/* Now do whatever rescaling of PTSIN values is required. */
/**********************************************************/

        if( opcode == V_UPDATE || opcode == V_CLEAR ||
            opcode == V_CLOSEWORK || opcode == V_CLOSEVWORK )
        {
            break;              /* exit display loop on any of these */
        }

        else if( opcode == SET_CLIP )
        {
            if( intin[0] == 0 )     /* Are we turning clipping off? */
            {
                clip[0] = glclip[0];    /* Reset clip rectangle */
                clip[1] = glclip[1];    /* to original */
                clip[2] = glclip[2];
                clip[3] = glclip[3];    /* And reset the clip flag so that */
                intin[0] = 1;           /* we still clip to our output window. */

                ptsin[0] = clip[0];     /* Put the adjusted clip */
                ptsin[1] = clip[1];     /* rectangle back into ptsin */
                ptsin[2] = clip[2];     /* And fall through to vdi() below */
                ptsin[3] = clip[3];
            }
            else
            {
                for( dummy = 0; dummy < 4; dummy += 2 )     /* Determine */
                {
                    xoffset_scale( &ptsin[dummy] );         /* the desired clip */
                    yoffset_scale( &ptsin[dummy+1] );       /* rect and */
                }                                           /* intersect with */
                intersect( glclip, ptsin );                 /* global clip rect */
            }
        }

        else if( opcode == GDP )
        {
            xoffset_scale( &ptsin[0] );
            yoffset_scale( &ptsin[1] );

            if( contrl[5] == ARC || contrl[5] == PIE )
            {
                xscale_only( &ptsin[6] );
            }

            else if( contrl[5] == CIRCLE )
            {
                xscale_only( &ptsin[4] );
            }

            else if( contrl[5] == ELLIPSE || contrl[5] == ELLARC ||
                     contrl[5] == ELLPIE )
            {
                xscale_only( &ptsin[2] );
                yscale_only( &ptsin[3] );
            }


            else if( contrl[5] == JUSTEXT )
            {
                xscale_only( &ptsin[2] );
            }

            else
            {
                for( dummy = 2; dummy < contrl[1]*2; dummy += 2 )
                {
                    xoffset_scale( &ptsin[dummy] );
                    yoffset_scale( &ptsin[dummy+1] );
                }
            }
        }

        else if( opcode == LINE_WIDTH || opcode == MARKER_HEIGHT ||
                 opcode == VST_HEIGHT )
        {
            xscale_only( &ptsin[0] );
            yscale_only( &ptsin[1] );
        }

        else if( opcode == VST_POINT )
        {
            points = (WORD)(((long)intin[0] * v_ptsize_factor)/1000);
            intin[0] = points;
        }

        else if( opcode == VST_ARBPOINT )
        {
            points = (WORD)(((long)intin[0] * v_ptsize_factor)/1000);
            intin[0] = points;
        }

        else if( opcode == VST_SETSIZE )
        {
            points = (WORD)(((long)intin[0] * h_ptsize_factor)/1000);
            intin[0] = points;
        }

        else if( opcode == ESCAPE )
        {
            if( contrl[5] == BITIMAGE )
            {
                xoffset_scale( &ptsin[0] );
                yoffset_scale( &ptsin[1] );
                xoffset_scale( &ptsin[2] );
                yoffset_scale( &ptsin[3] );

/* If we wanted to support this call for the screen, we could */
/* plug in a call to our own display IMG file routine here */

                if( dev_handle == scrnhndl )
                {
                    /* display IMG on screen */
                }
            }

/* If not v_bit_image(), then don't know what it is. */
/* So scale and offset anything in ptsin[] and pass call through to vdi() */

            else
            {
                for( dummy = 0; dummy < contrl[1]*2; dummy += 2 )
                {
                    xoffset_scale( &ptsin[dummy] );
                    yoffset_scale( &ptsin[dummy+1] );
                }
            }
        }

/* It's either not a special case, or we don't know what kind */
/* of special case it is. Scale & offset the coordinates in ptsin. */
/* That may not be right for an unknown special case function, */
/* but doing nothing to the ptsin[] values is almost certainly wrong. */

        else
        {
            for( dummy = 0; dummy < contrl[1]*2; dummy += 2 )
            {
                xoffset_scale( &ptsin[dummy] );
                yoffset_scale( &ptsin[dummy+1] );
            }
        }

/* stuff d0 & d1 with magic number and parameter */
/* block address, then execute VDI command */

#ifdef LATTICE
        vdi( vdipb );
#else
        vdi();
#endif

    }
    while( bufpos < endbuf );       /* gimme a 'break' ok? */
}
```

```
/*********************************************/
/* meta_info() returns page size in microns, */
/* page size in metafile coordinates         */
/*********************************************/

void
meta_info( metafile, header )
WORD *metafile;
Metafile_Info *header;
{
    bufpos = metafile;      /* Set up buffer ptr used by get_wd() */

    header->magic = get_wd();
    header->hdrlen = get_wd();
    header->version = get_wd();
    header->ndcflag = get_wd();

    header->minx = get_wd();
    header->miny = get_wd();
    header->maxx = get_wd();
    header->maxy = get_wd();

    header->pg_wid = get_wd();  /* Page width in microns */
    header->pg_ht = get_wd();   /* Page height in microns */

    header->llx = get_wd();     /* Lower left coordinate of page */
    header->lly = get_wd();     /* Lower left coordinate of page */
    header->urx = get_wd();     /* Upper right coordinate of page */
    header->ury = get_wd();     /* Upper right coordinate of page */

    header->wc = header->urx - header->llx;
    header->hc = header->lly - header->ury;
}

/****************************************************************************/
/* Calculate intersection of two rectangles, and return rectangle of the    */
/* intersecting area.  Returns '1' if rectangles intersect, '0' if not.     */
/* If no intersection, returned rectangle is undefined.                     */
/****************************************************************************/

WORD
intersect( a, b )
WORD *a, *b;
{
WORD x, y, w, h;

    x = max( a[0], b[0] );
    y = max( a[1], b[1] );
    w = min( a[2], b[2] );
    h = min( a[3], b[3] );

    b[0] = x;
    b[1] = y;
    b[2] = w;
    b[3] = h;

    return( (WORD)((w > x) && (h > y)) );
}

/****************************************************************************/
/* Get word or long out of buffer that is not necessarily word-aligned.     */
/****************************************************************************/

WORD
get_wd()
{
register char *bufptr;
union {
    char    byt[2];
    WORD    wd;
} theWord;

    bufptr = (char *)bufpos;
    theWord.byt[1] = *(bufptr++);
    theWord.byt[0] = *(bufptr++);
    bufpos = (WORD *)bufptr;

    return( theWord.wd );
}
```

```
/*******************************************************/
/* Here are the coordinate scale and offset functions. */
/*******************************************************/

void
xoffset_scale( xval )
short *xval;
{
register long xv;

    xv = (long)*xval + x_offset;
    xv = xv * world_w;
    xv /= area_wd;
    xv &= 0xffff;
    xv += world_x;

    *xval = (short)xv;
}


void
yoffset_scale( yval )
short *yval;
{
register long yv;

    yv = (long)*yval + y_offset;
    yv = yv * world_h;
    yv /= area_ht;
    yv &= 0xffff;
    yv += world_y;

    *yval = (short)yv;
}

void
xscale_only( xval )
short *xval;
{
register long xv;


    xv = (long)*xval;
    xv = xv * world_w;
    xv /= area_wd;

    *xval = (short)xv;
}

void
yscale_only( yval )
short *yval;
{
register long yv;

    yv = (long)*yval;
    yv = yv * world_h;
    yv /= area_ht;

    *yval = (short)y
}
```

Listing #3

```
/*  Code for this is:

    move.l  (sp),d1
    moveq.l #$73,d0
    trap    #2
*/

#pragma inline d0=vdi() { register d0, d1, d2, a0, a1, a2; "221770734e42"; }
```

Atari ST/TT Q & A

Mike Fulton

Q: I want my application to save out color GEM IMG picture files. I already know how to save out monochrome IMG files, how are color IMG files different?

A: The main differences are that the IMG file header indicates more than one bitplane, and that the compressed image data for each scanline now contains data for more than one bitplane.

Basically, you go through the same steps as for monochrome IMG files, except that immediately prior to compressing a scanline, you must convert it from device-specific format into VDI-standard format using the GEM VDI ***vr_trnfm()*** function. This gives you a VDI-standard format raster form for the scanline, which you then compress and save in the same way you would for a monochrome IMG file, except now there is more data.

Converting the image data to VDI standard format is done individually for each scanline, resulting in a file structured as shown in figure #1. Some programs are known to incorrectly convert the entire picture at once, resulting in a file structured as shown in figure #2. (In both figures, each square represents part of the compressed picture data.) Carefully read Appendix I of the GEM VDI manual, and you will note that the mention of how the bitplane data is encoded is given within the context of how a single scanline item is encoded, not the overall picture.

Doing the conversion scanline by scanline makes it easier for GEM printer drivers to print IMG files, since they can get all the information for a single scanline all at once without having to jump back and forth through the file. They can start at the beginning of the IMG file grab a few scanlines, scale them and print them as required, then go on to the next few scanlines, again and again until the entire picture is done. (Imagine the printer driver reading the data from figure #1 from left to right, top to bottom, straight through from start to finish.)

If the whole picture was converted in one step, the printer driver would have to decompress almost the entire picture to get all the data needed for a scanline. This is because the data compression makes it impossible to predict where the data for each bitplane is, and the only way to find it would be to decompress everything until you found what y ou wanted, even if you were doing scanline 0.

Doing the conversion scanline by scanline also requires a much smaller memory buffer to convert the picture.

Note: Appendix I of the GEM VDI manual also refers to the planes as the "red" plane, the "green" plane, the "blue" plane, and the "grey" plane. This refers to the way color data is stored on an EGA graphics card for a PC, and is an example only, with no meaning beyond that. (If you've read the book ***Graphics File Formats***, please note that its description of how color information is stored in an IMG file seems to be based on an misinterpretion of that example, and is completely incorrect.)

As far as color palette information goes, there is no officially sanctioned way of saving color palette information in an IMG file. However, there is an unofficial extension to the IMG format known as XIMG which adds the color palette information to the end of the file header in the following way:

```
WORD 1 = new header length in WORDs
WORD 8 = $5849
WORD 9 = $4d7d   (Word 8 & 9 = "XIMG")
WORD 10 = 0
WORDs 11-?  VDI-style RGB information
(3 WORDs per color pen, values 0-1000)
```

Note: Some programs using the XIMG format are known to incorrectly convert the image data as mentioned earlier.

Scanline #0	Plane 0	Plane 1	Plane 2	Plane 3
Scanline #1	Plane 0	Plane 1	Plane 2	Plane 3
Scanline #2	Plane 0	Plane 1	Plane 2	Plane 3
Scanline #3	Plane 0	Plane 1	Plane 2	Plane 3

Figure #1

Plane #0	Scanline 0	Scanline 1	Scanline 2	Scanline 3
Plane #1	Scanline 0	Scanline 1	Scanline 2	Scanline 3
Plane #2	Scanline 0	Scanline 1	Scanline 2	Scanline 3
Plane #3	Scanline 0	Scanline 1	Scanline 2	Scanline 3

Figure #2

Q: How do I have my program look for an event on either mouse button using ***evnt_multi()*** or ***evnt_button()***?

A: Use an *ev_mbclicks* value of 0x101, a *ev_mbstate* value of 0, and an *ev_mbmask* value of with the ***evnt_multi()*** and ***evnt_button()*** functions and you will get an event whenever either mouse button is pressed. This method has worked in all versions of TOS and is now official. (This method does not work for detecting mouse button releases.)

(Portfolio Q & A, Continued from Page 4)

DTMF

Q: How do I vary the amount of time the Portfolio can generate tones and the amount of space between the tones?

A: The technical reference guide was incomplete in its description of this call, so here is a short programming example in assembly which changes the DTMF duration.

```
; DTMF duration testing under
; assembled under TASM

; 2710h(10000) short duration,
; 72e3h(29411) = default duration

    .model small
    .stack 100h
    .data

dtmf_dur    dw  0
temp        dw  0
numstr      db  '00001111',0

    .code
    mov     ax, @data
    mov     ds, ax

    mov     ah, 18h       ; Mute States call
    mov     al, 08h       ; Get DTMF duration
    int     61h

    mov     dtmf_dur, dx  ; save old duration value
                          ; from register

    mov     ah, 18h       ; Mute States call
    mov     al, 09h       ; Set DTMF duration
    mov     dx, 2710h     ; stuff new duration value
    int     61h           ; into register

    mov     ah, 17h       ; dial number to check tone
    mov     temp, @data   ; duration
    mov     ds, temp
    mov     si, OFFSET numstr
    mov     cx, 8h        ; length of string = 8 characters
    int     61h

    mov     dx, dtmf_dur     ; stuff old duration value
                             ; into register
    mov     ah,18h           ; Set DTMF duration
    mov     al,09h
    int     61h

    mov     ah, 17h          ; dial number to recheck
                             ; old tone duration
    mov     temp, @data
    mov     ds, temp
    mov     si, OFFSET numstr
    mov     cx, 8h
    int     61h

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;terminate

    mov     ah,4ch
    int     21h

end
```

Atari Computer Corporation
1196 Borregas Ave.
Sunnyvale, CA 94089-1302

ATARI.RSC
The Developer's Resource